PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 17/30

A1

(11) International Publication Number: WO 97/15018

(43) International Publication Date: 24 April 1997 (24.04.97)

(21) International Application Number: PCT/US96/15620

(22) International Filing Date: 26 September 1996 (26.09.96)

(30) Priority Data:
08/543,644 16 October 1995 (16.10.95) US

(81) Designated States: CA, CN, JP, KR, European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

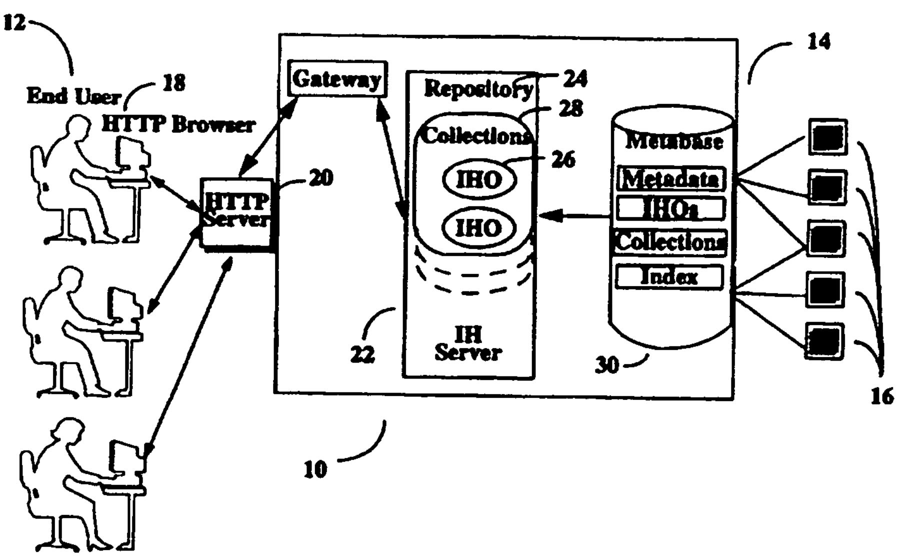
Published
With international search report.

(71) Applicant: BELL COMMUNICATIONS RESEARCH, INC. [US/US]; 445 South Street, Morristown, NJ 07960-6438 (US).

(72) Inventors: MARCUS, Howard; 12 Harrison Street, Edison, NJ 08817 (US). SHAH, Kshitij, Jawahar; 119 Jeremy Court, Edison, NJ 08817 (US). SHETH, Amit, Pravinkumar; 1140 Laurel Pointe, Bogart, GA 30622-2856 (US). SHKLAR, Leon, A.; 160 Stults Lane, East Brunswick, NJ 08816 (US). SURAK, Jerome, Raymond; 277 Mohawk Trail, Bridgewater, NJ 08807 (US). THATTE, Satish, Mukund; 18 Crestview Drive, Kendall Park, NJ 08824 (US).

(74) Agents: WHITE, Lionel, N. et al.; International Coordinator, Room 1G112R, 445 South Street, Morristown, NJ 07960-6438 (US).

(54) Title: METHOD AND SYSTEM FOR PROVIDING UNIFORM ACCESS TO HETEROGENEOUS INFORMATION



(57) Abstract

Our invention is a system and methodology for integrating heterogeneous information in a distributed environment by encapsulating data about existing and new information into objects (16). The process of encapsulating the information requires extracting from the information metadata. Creating from the metadata, a database (30), where the metadata is grouped into objects (26) and groups of objects (28) which are logically associated into collections (28). This database of object and collections is instantiated into runtime memory of a server (22), organized into repositories (24) of objects (20) and collections (28). A user (12) seeking access to the information would then, using an HTTP compliant browser (20), access the server (22) to access the information through the objects (26) created and stored in the server.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AM	Armenia	GB	United Kingdom	MW	Malawi
AT	Austria	GE	Georgia	MX	Mexico
AU	Australia	GN	Guinea	NE	Niger
BB	Barbados	GR	Greece	NL	Netherlands
BE	Belgium	HU	Hungary	NO	Norway
BF	Burkina Faso	IE	Ireland	NZ	New Zealand
BG	Bulgaria	IT	Italy	PL	Poland
BJ	Benin	JP	Japan	PT	Portugal
BR	Brazil	KE	Kenya	RO	Romania
BY	Belarus	KG	Kyrgystan	RU	Russian Federation
CA	Canada	KP	Democratic People's Republic of Korea	SD	Sudan
CF	Central African Republic			SE	Sweden
CG	Congo	KR	Republic of Korea	SG	Singapore
CH	Switzerland	KZ	Kazakhstan	SI	Slovenia
CI	Côte d'Ivoire	LI	Liechtenstein	SK	Slovakia
CM	Cameroon	LK	Sri Lanka	SN	Senegal
CN	China	LR	Liberia	SZ	Swaziland
CS	Czechoslovakia	LT	Lithuania	TD	Chad
CZ	Czech Republic	LU	Luxembourg	TG	Togo
DE	Germany	LV	Latvia	TJ	Tajikistan
DK	Denmark	MC	Monaco	TT	Trinidad and Tobago
EE	Estonia	MD	Republic of Moldova	UA	Ukraine
ES	Spain	MG	Madagascar	UG	Uganda
FI	Finland	ML	Mali	US	United States of America
FR	France	MN	Mongolia	UZ	Uzbekistan
GA	Gabon	MR	Mauritania	VN	Viet Nam

METHOD AND SYSTEM FOR PROVIDING UNIFORM ACCESS TO HETEROGENEOUS INFORMATION

TECHNICAL FIELD OF THE INVENTION

This invention relates to data processing systems and networks. More specifically, this invention relates to methods and systems for accessing distributed heterogeneous information sources and databases.

BACKGROUND OF THE INVENTION

Given the advances in modern computer technology and the proliferation of relatively inexpensive off-the-shelf authoring and office automation software, the ability to create information has increased dramatically. Naturally therefore, the size, diversity, and quantity of information repositories have also increased. As a result, enormous amounts of information have been accumulated within corporations, government organizations, and universities. With the advances in data communication technology and computer networking, much of this information is in electronic repositories on networks accessible to anyone with a computer connected to these networks. However, the information is heterogeneous; i.e. stored in many forms of differing types and representations.

In such an environment in order for users to access these heterogeneous types of information, they not only have to know the about the existence and location of the information but also the format of the information, the different database query languages procedures, and differing access and retrieval procedures for accessing and retrieving this information.

Accordingly, knowledge workers are spending too much time trying to locate, access and retrieve the information they need. Often times because of these barriers to access, knowledge worker's give up trying to access the information and recreate the same information in another repository in yet another inconsistent manner. These problems in accessing heterogeneous information reduce individual and organization productivity, thereby increasing the cost of doing business.

To address these problems, those practicing in the art have attempted to build uniform information repositories by relocating and reformatting the original information in some standard format and at centralized locations. This approach requires the design and maintenance of an ever-increasing number of ever-changing format translators. In addition, the initial conversion of the information often requires substantial human and computing resources. Furthermore, maintaining the repositories requires either creating new and updating information in the uniform format, or continuously managing changing data in different formats. These approaches are not only resource intensive, but because they are based on a centralized model of system management, they are characterized by a performance, administrative and reliability bottleneck, inherent in centralized systems.

Another problem presented by the prior art is that sophisticated indexing and search techniques are available only for certain types of information, or such techniques come embedded within an application and cannot be applied to other

kinds of information, i.e., such techniques are part of a closed system. Therefore, on a network with heterogeneous information, users are therefore burdened with having to cope with multiple indexing and search techniques that are developed and applied in idiosyncratic ways to handle different kinds of information.

One recent advance in the art of providing users easy access to information from a variety of sources is the development of the World Wide Web on the Internet. The users, using hypertext transfer protocol (HTTP) browsers, connecting to HTTP servers have access to numerous sources of information. The information they are be able to retrieve are textual files formatted using a hypertext markup language (HTML). These HTML files not only provide users with textual information, but embedded within the text are pointers to other sources of information, they may be graphic, audio, video or textual. Most commercially available browsers (e.g. Mosaic, Netscape) contain tools capable of displaying graphic or textual information. However, in order for this information to be displayed it must have been at some point converted into HTML files. However, there is a tremendous amount of legacy information in networks that could be made available to users if there was a means to access it without the owners or providers of the information having to convert it to HTML files.

Accordingly, what is needed in the art is a system and method for providing users with integrated access to large amounts of heterogeneous information without the end-user needing to know the type, format or location of the information

and without burdening the owners or providers of the information with having to translate, relocate or re-format the information.

SUMMARY OF THE INVENTION

It is therefore an object of the present invention to provide users with integrated access to large amounts of heterogeneous information without the end-user needing to know the type, format or location of the information. It is a further object of the present invention to accomplish these goals without having to burden the information owners with having to translate, relocate or reformat the information. These objectives are achieved and an advance in the art is made by our invention. Our invention is a system and methodology for integrating heterogenous information in a distributed environment by encapsulating data about existing and new information in objects without converting, restructuring, or reformatting the information. The process of encapsulating the information requires extracting from the information metadata. Creating from the metadata, a database, where the metadata is grouped into objects and groups of objects are which logically associated into collections. This database of object and collections is instantiated into runtime memory of a server, organized into repositories of objects and collections. A user seeking access to the information would then, using an HTTP compliant browser, access the server to access the information through the objects created and stored in the server. Our invention provides an integrated view of and access to diverse

heterogeneous information. Our invention also provides tools for accessing, retrieving, browsing and administering the information.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 illustrates a system in accordance with one embodiment of our invention.

Figure 2 depicts a method for pre-processing the information units in accordance with one embodiment of our invention.

Figure 3 depicts a method for accessing heterogeneous information in accordance with one embodiment of our invention.

Figure 4(a) depicts the format a of the metadata as used in the present embodiment of our invention

Figure 4(b) depicts a table defining the metadata fields as used in the metadata format of the present embodiment of our invention.

Figure 5 depicts the format of one embodiment of an object identifier as used in our invention.

Figure 6 depicts a table that defines the attributes of an ihMeta object.

Figure 7 depicts a class inheritance diagram for the ihArtifact family of classes as defined for the present embodiment or our invention.

Figure 8 is a table of ihArtifact classes as used in the present embodiment of our invention.

Figure 9 is a table of ihArtifact sub-class definitions as used in the present embodiment of our invention.

Figure 10 is a table the defines ihGraph class as used in the present embodiment of our invention.

Figure 11 illustrates the relationship between the run-time modules operating in a server in accordance with our invention.

Figure 12 depicts an example interaction diagram of the operating in accordance with the present embodiment of our invention.

Figure 13 illustrates the ih_prep process and extractors and indexers in accordance with the present invention.

Figure 14 illustrates the process for conducting metadata context queries in accordance with the present embodiment of our invention.

Figure 15 illustrates the process for conducting information content queries in accordance with the present embodiment of our invention.

Figure 16 illustrates the process for invoking a server side browser in accordance with the present embodiment of our invention.

Figure 17 illustrates the process for invoking a client side browser in accordance with the present embodiment of our invention.

DETAILED DESCRIPTION

Described below is one preferred implementation of the present invention which is illustrated in the accompanying drawings. This one embodiment or our invention is described as it has been implemented in a product, known as the InfoHarness™ software and system. This description of our invention is organized into six sections. First, we define terms that will be

™ InforHarness is a trademark of Bell Communications Research Inc., the assignee of this patent.

used throughout the specification. Second, we provide a high level overview of our system and method. Thirdly, we describe in detail the process for metabase preparation. In the fourth section, we describe the operation of a gateway, necessary in our embodiment, to connect the HTTP server to the InfoHarness Server (It isn't material to our invention to have this gateway as a stand alone process but could in other embodiments be embedded within the server). In the fifth section, we describe the operation of the InfoHarness server which operates in accordance with our invention. Finally in the sixth section, we describe the interactions between the components of our inventive system. These descriptions are only exemplary of the invention. The present invention is not limited to the implementations described, but may be realized by other implementations.

A. DEFINITIONS

An information unit, or IU, is a piece of information that may be of interest to an end user. The most common kind of IU is a document stored in a single file. An IU can also represent a portion of a file (such as a single program function in the C language within a larger source code file, or a single email message in an email file, etc.), a grouping of many files, or other kinds of information.

Metabase is a file or database of metadata extracted from the information units and organized into InfoHarness Objects and collections.

Metadata is "data about data" -- it is data that describes

various salient characteristics of some other data. For instance, metadata about this patent specification could include its filing date, the inventors names, a keyword summary, etc.

An InfoHarness Object, or IHO, is an encapsulation of an information unit that is be accessed using our inventive system. An IHO encapsulates metadata describing the salient characteristics of an IU.

A collection represents a set of IHOs. Collections are logical entities; that is, the information units encapsulated by the member IHOs do not have to be physically co-located in the same directory. Encapsulated files can be distributed on many systems of a network. Further, IHOs can be members of more than one collection. Collections can be nested (i.e., contain other collections). They can also be indexed or non-indexed (e.g. processed to permit content searches). Collections, thus, provide a logical view of physically distributed, heterogeneous information.

A repository is the a of collections. Its contents are accessed through an InfoHarness server operating in accordance with our invention.

A gateway is a component of the present embodiment of our of our invention that provides an means for connecting an Hypertext Transfer Protocol (hereinafter HTTP) server to an InfoHarness server according to the Common Gateway Interface (CGI) specification which is well known by those who practice in the art.

An InfoHarness Server (IH server), is a server operating in accordance with our invention.

B. OVERVIEW

One embodiment of our invention is illustrated in Figure 1. Our inventive system 10 is interposed between a plurality of end users 12 who want access to heterogeneous information 14 composed of a plurality of IUs 16. In the embodiment described herein, the end users 12 use an HTTP-compliant browser 18 to connect to an HTTP server 20, which in turn connects to an IH server 22. Within the IH server 22 instantiated into memory is a respository 24 of IHOs 26 and collections 28. This respository 24 was created from a database 30 of IHOs and collections created from metatdata extracted from the IUs 16 and stored in a database. Users 12 connected to the IH Server 22 then can obtain IHO, metadata or search collections, using any user-specified criteria to retrieve the target information from the IUs 16.

Our inventive method is composed of two phases. Phase 1 is the registration phase under which IU's are pre-processed to create IHO's, collections and repositories. Phase 2 is the information access phase wherein end-users access the IH server through the HTTP server and use the IHO's, created in the registration phase and loaded in the memory of the IH server, to locate and access the IU's.

Figure 2 illustrates the methodology embodied within the registration phase. An information provider or InfoHarness administrator having information which the provider desires to

make accessible to users, would invoke an InforHarness registration procedure (software) to register the information units 30. Upon invoking the InfoHarness registration procedure, the administrator would first invoke a pre-processor 32 to prepare the information for the extraction process. The next step involves the administrator invoking one of a plurality of extraction processes 33 to extract metadata from the information units that are be registered (the appropriate extractor process depends on the type of information the administrator is registering). The output of the extraction process is the creation of a metabase (which is a file or database) of IHOs 34. This metabase contains metadata of the information units logically collected into a collection, and also information about IHOs and collections and the relationships among them. Upon the creation of the metabase the registration phase is complete.

Figure 3 illustrates the methodology for accessing the information in accordance with our invention. First the IH server must be initialized, then the IHOs are loaded from the metabase into the server's memory and organized into repositories 36. After the server is initialized and running, the IH server enters a main event loop and waits for requests from clients 38. End-users then access the IH server through an HTTP server 40. Once the end-users access the IH server, they perform one of three actions to select an object 42: (1) a metadata based query, (2) a content based query, or (3) explicitly navigate around the IHOs. Once an object is

selected, it can be accessed and browsed by activating either a client side browser 44 or server side browser 46. The user may also operate on the object choosing from a set of procedures such as print, store, fax, etc.

C. REGISTRATION PROCESS

As described above, the registration process involves an owner, creator, or provider of information working with a system administrator to pre-processing the information for the purposes of extracting metatdata in a format usable by our IH server. Registration is accomplished in four steps: pre-processing the physical data, extracting the metadata, storing the metadata in a metabase, and transferring the extracted metadata from the metabase to the IH server.

The main function of the pre-processing is to process the physical data and build logical structures (IHOs and collections) which the IH server can later use for presentation to end users. Physical data, in this sense, includes formatted, unformatted, structured, and unstructured data. It could also be dynamic; e.g., SQL queries or newsfeeds.

Metadata, which is extracted in accordance with the methods described herein can be content-dependent, content-descriptive, or content-independent. Content-dependent metadata is based strictly on the contents of the physical data. Examples of content-dependent metadata are keyword indices for textual data, grids for image data, speaker change lists for audio data, etc. As the name "content-descriptive" suggests, it describes the physical contents of the data. Examples include spatial

information for video data, the subject of a talk for audio data, document composition for multimedia data, etc. Content-independent metadata, on the other hand, does not rely on the specific content of the underlying data for it's values. Media type, document history and location, temporal information for video and audio, etc., are examples of content-independent metadata.

In accordance with this embodiment of our invention, all data to be registered with the system must be accessible as a file on a mounted file system. This typically means that the data must all be on the same LAN, although it may be stored on multiple file servers if those servers are directly accessible, such as via an network file server (NFS) mount.

Related data, although physically scattered, is usually grouped together in some logical structure superimposed on the underlying physical data. A directory structure is one example of a logical structure which usually exists for most file systems. Also, the system administrator working with the InfoHarness application builder may determine that users might be interested in relationships among collections and IHOs. As and example, a parent-child relationship between the two collections could be imposed. Other relationships that could be modeled are: 'contains,' 'is contained in,' and 'part-of.'

The end result of pre-processing and metadata extraction is the creation of a metabase (which may be a file or database) containing metadata, IHOs and collections. As described above, when this information is loaded from the metabase into an IH

server, it is materialized as IHOs and collections in the server's memory organized into repositories. In our invention, we use different extractor processes for various document data types. These extractor processes are easily created using skills well known in that art. As an example, we use extractors for Text, PostScript, HTML, man pages, and e-mail message files.

An IHO encapsulates a single IU. A collection does not encapsulate any IU, rather it is a set of other IHOs or collections. An IHO, encapsulating an IU, would thus have a unique identifier by which to distinguish itself from other IHOs.

A collection, is a set of IHOs, related together at the discretion of the system administrator of InfoHarness application builder. Physically, in the embodiment described herein, a collection is represented by a number of Unix files in a common subdirectory, whose name is the name of the collection. This collection directory contains several important files:

IH_SUMMARY file -- This file contains some meta-information about the collection itself, such as where the index is located (if any), what the collection metadata filename is called, etc.

Metadata file -- A file that contains the metadata extracted during the registration process.

Index -- Depending upon the indexing scheme used, one or more index files may be present.

Our metadata extraction processes are summarized by the following pseudocode:

1. Validate user supplied options for the extraction process.

```
2.  For each directory to be scanned
            For each file eligible for extraction
                   Invoke extractor
                   Process returned metadata
                   Collect extracted text

3.  Index extracted text if requested
```

Pre-processing consists of extracting metadata from physical information sources, creating representations for IHOs, collections and relationships, and optionally creating an index on textual contents of the sources. In the current embodiment, the physical information sources should exist on the same file system as the pre-processor and indexer. The metadata is also stored on this file system at the location specified by the administrator.

The pre-processor uses extractor methods for the extraction of metadata from the physical information sources. These are type-specific methods which process the information sources and return metadata in a specific format. In our embodiment, the preprocessor does not analyze the source type to invoke an extractor; instead the system administrator of our IH server is expected to indicate a particular extractor which will then be used for metadata extraction. The pre-processor treats all the IHOs generated as constituents of a collection. A user-specified location is used to store the metadata files created. The user has the option to append newly generated metadata to an existing collection.

The user can also indicate whether this generated collection should have a text index built for it, and if so, which indexing technology to use for this purpose. The indexing

technology itself is not part of the present invention. However, the architecture of the present invention allows a variety of indexing technologies to be used in a plug-and-play fashion. Example indexing technologies are WAIS and GLIMPSE. If an index is generated, it is installed in the same directory as the metadata files. A cross-reference file is also generated which maps the index database objects to the to IHOs. If indexing is not performed the generated collection is treated as a set.

A typical extractor takes as input the location of the information source which is to be encapsulated. It returns a formatted string which the pre-processor interprets to generate metadata entries that are stored in a metadata file. The metadata file itself has a well-defined format, described in more detail below. The extractor also extracts the text associated with the generated IHOs. To extract text from a 'C' file, for instance, the C file has to be parsed to recognize comments and function signatures, because indexing the language constructs and variable names does not usually make sense. In this case an IU would be associated with either a function or the file as a whole. Representative information is also extracted and associated with the IU. This will be displayed to the user at browse time; e.g., for mail messages the subject line is used as a representative, for HTML documents the contents of the TITLE construct are used, etc.

Metadata is passed from the extractor in a format called the metadata transfer format. This format (a Perl data

structure) has constructs which allow arbitrary graph structures to be imposed on top of the IHOs (e.g., parent-child relationships between collections). The object's type and subtype are associated with the IUs and are both determined by the extractor process. Finally, the location attribute (i.e., a value used to locate the IU in the file system) is also determined by the extractor. This could be the full path for a UNIX file for cases where the IU is associated with the whole file. It could also be a Uniform Resource Locator (URL) (as it is understood on the World Wide Web) or some other locator. URLs are used for HTML documents. The location of a 'C' function, on the other hand, could be specified as 'filename%function_name'. There is no requirement on the precise format of this locator, as long as the browsing methods can decipher it to retrieve the original data associated with that IU. Besides these, any number of attribute-value pairs can be associated with the IU. e.g. the attribute name associated with an IU will contain the representative information extracted by the extractor. For IHOs which do not contain an IU any arbitrary text could be assigned to this attribute; e.g., for a collection IHO the name of the collection can be assigned and this will be displayed to the user.

Metadata is transferred between the extractors and the pre-processor as a structured Perl string, whose format is shown as 48 in Figure 4(a). Each IU has six fields of metadata associated with it (e.g., f11 through f16), each separated by a colon, and each IU's metadata is separated from the next IU's by

a vertical bar 52. Figure 4(b) depicts a table 54 that summarizes the purpose of each field.

The location field 55 is created by the extractor process to identify where the IU is stored.

The Unique ObjId Indicator field 56 instructs the pre-processor whether to use the Location to construct a unique object identifier. For some cases the extractor supplied locator is guaranteed to be unique so that the pre-processor need not manipulate it. One such case is IUs associated with HTML files, for which URLS are generated by the extractor as IU locations. These URLs are unique. If this flag is set, the pre-processor constructs a unique identifier for the object.

The ordinal value of the Depth field 58 indicates the depth of that IU in an in-order traversal of the desired repository structure. The collection object, which is the root of this tree, is pre-assigned a depth of 0. An extractor returning a simple list of file IUs that are to be a part of this collection would assign a depth of 1 to each of these file IUs. The pre-processor then makes all of these file IUs children of the collection object. An example of the structure in the metadata transfer format is shown in Fig. 5.

The Subtype field 60 is determined by the extractor and is used later by the IH server to determine how to access the actual IU.

The Subject field 62 contains summary information related to an IU and is what the user will see as the "name" of the object at the time of browsing.

The last field 64 is the text body of the IU, to be used if the collection is being indexed.

The sequence of the entries in this metadata transfer format stream 48, along with the value of its depth field 58, determines it's position in the collection structure built by the pre-processor. An IU may be represented multiple times in this stream, possibly to assert a relationship with other IUs, but a metadata entry is made for only the first occurrence. An empty text field indicates that the IU need not be cross-referenced for indexing.

Since the colon (':') and vertical bar ('|') characters are used as delimiters for the fields and IU entries, respectively, they need to be "escaped" with a backslash ('\') if they occur anywhere within the content of any of the fields.

After the pre-processor has parsed the stream returned by the extractors it stores the object representations into a single metadata file. These metadata entities will be read in by the IH server when it is brought up and instantiated as in-memory IHO representations. There is a fixed structure to the entries appearing in the metadata files.

There are two kinds of entries in the metadata file, object entries and relationship entries. Object entries are flat representations of the IHOs whereas relationship entries represent parent-child relationships between IHOs.

Object entries have an object identifier. This object identifier could be constructed by the pre-processor or the extractor as specified by the indicator in the metadata

transfer format. If the pre-processor constructs the object identifier, it does so in a specific format. The format is:

machineid:location:subtype

The machineid is a unique physical machine identifier of the machine on which the pre-processor is run. This field is automatically generated by the pre-processor. The location and subtype field values are assigned based on the values returned in the metadata transfer stream. The location field, for a simple or composite IHO would be the location of the associated IU. For a collection IHO this would be the location of the collection; e.g., for an indexed collection it would be the location of the index. The subtype field value is the same as the subtype value returned in the metadata transfer stream. For a collection IHO this is the index type; i.e., wais or glimpse.

An object entry is of the form as shown in Figure 5. The first field 70 serves as the object identifier. This object identifier is used for uniquely identifying the object and serves as a key. The type 71 and subtype 72 values correspond to non-terminal and terminal classes in the server abstract class hierarchy. The location value 73 is used by the browser methods to retrieve the data associated with the IU encapsulated by this object. Following this there could be an arbitrary number of attribute-value pairs 74. The 'name=string' pair is used when the user is browsing the repository. The string is displayed to the user.

A relationship entry is of the form: [objid1 | objid2]

This establishes a parent-child relationship between the objects represented by objid1 and objid2, with the former being treated as a parent of the latter.

There are no constraints on the order in which entries appear in the metadata file except that the object entry has to appear before its object identifier can take part in a relationship.

D. GATEWAY PROCESS

In our illustrative embodiment, the HTTP server is connected to the IH server through a gateway. This gateway interacts with two types of programs: an HTTP server (which in turn interacts with an HTTP browser (e.g. Mosaic or Netscape). Any HTTP-compliant browser can interact with the HTTP server. and the IH servers. There are five actions exported by the gateway to the HTTP browser. They are: Setup, Init, Expand, Query, and Show. There are four actions exported by the IH server that the gateway uses. They are: Init, Expand, Query, Show

By design, the HTTP protocol is stateless (see http://info.cern.ch/hypertext/WWW/Protocols/HTTP/HTTP2.html for information). This implies that interactions between an HTTP browser and any HTTP server is stateless. No information about clients is kept by the HTTP server between connections. This is contrary to the needs of many applications, including our gateway. To understand why this is so, consider the information necessary for a user to issue a content-based query against a collection. The user must specify: the machine where the IH server they wish to interact with is running, the port number the IH server is using to accept connections, their X display value, the query text that should be used to select objects from

the collection, the maximum number of hits to return on a successful query, and the collection against which the query will be executed. One approach to gathering this information would be to force the user to specify all necessary parameters by hand on each interaction with the gateway. However, that would clearly not be a very user-friendly approach. Instead, our design is such that the user only needs to enter certain information once, on a "setup" screen. All screens that are presented to the user after the setup screen have "state" information embedded into the URLs, so that if the user activates the URL link, the embedded state information can be extracted from it. One side effect of this is that, since some of the HTML pages created by the gateway have many URLs, and each of these URLs contains all of the information necessary to maintain the state of the user's interactions, there is a large amount of duplicated information in the URLs on a single page.

This arrangement causes the gateway to spend time performing two tasks: retrieving information from incoming URLs, and reformatting the output of the IH server into URLs (and HTML).

Given the fact that interaction between the HTTP browser and the HTTP server is stateless, it does not necessarily make sense to talk about a correct sequence of calls to the HTTP server. As long as the HTTP browser passes valid requests to the gateway, the requests will be processed without regard to order. However, in order to develop a basic understanding of how the HTTP browser and the HTTP server interact, consider the

following sequence of events which many users will find typical.

The HTTP browser opens a URL pointing to the gateway (e.g., http://http.ctt.bellcore.com/cgi-bin/nph-ih.cgi). The HTTP server responds by returning the setup screen to the HTTP browser. The user determines the IH server to connect to and enters the correct information on the fill out form on the setup screen. Once the form is submitted, the gateway connects to the specified IH server and requests a list of collections managed by the IH server. For each item in the list returned by the IH server, the gateway generates a URL containing all the necessary information required to access this collection on the next interaction, and returns the list to the HTTP server, which in passes it to the requesting HTTP browser. The user can then select one of the collections returned by the gateway for further interrogation. If the collection is indexed, the gateway presents a form to the user for entering the search text. If the collection is not indexed, the gateway connects to the appropriate IH server (as specified in the URL) and requests the contents of the list. The list contents are then formatted appropriately in HTML by the gateway, and URLs are generated for each item in the list.

If the HTTP browser receives a fill out form, a search can be initiated. If the user submits a query, the gateway sends that request to the IH server. The IH server response is similar to the results returned when the members of a list are requested, and again, the gateway formats the results into a

list with their corresponding URLs. In either the search results list, or the simple list, the HTTP browser can select any of the items in the list. If the user selects an item (i.e., clicks on the link), this translates to saying "show me this item." The gateway contacts the appropriate IH server (again determined by the state information embedded within the URL) and requests the particular item. If the item has been designated as displayable by the IH server, the IH server retrieves the item and uses X to display the item back to the user. If the item has been designated as displayable by the HTTP browser, the IH server retrieves the item and sends it back to the gateway. The gateway determines (based upon the type of data returned) what Multimedia Internet Mail Extension (MIME) type the item corresponds to and returns the appropriate header information as well as the actual data to the HTTP browser.

Although IH users will find the steps outlined in the previous paragraph familiar, it is important to remember that these steps can occur in any sequence as long as the appropriate information is passed to the gateway. Again, the reason for this is the stateless nature of the HTTP. Some users may wish to exploit this feature. A user may wish to construct several "canned" queries against a particular IH server. The URL's representing these queries can be imbedded in other HTML documents providing more descriptive text regarding the queries, or their intended results. Another user may want to provide access to individual objects held by the IH server. They may construct URLs that point directly to the objects (even

objects that are members of an indexed collection) and circumvent the need for search queries to retrieve the objects.

The processing that occurs at the gateway is relatively straightforward. When an IH server generated link is activated by the user (e.g., the user clicks on an object on the query results screen), the gateway examines the URL that was activated. All such URLs are unescaped and validated. Unescaping a URL consists of replacing all sequences of the form %XX (where X is a valid hexadecimal value) with their corresponding ASCII value. Validating a URL consists of extracting the information contained in the URL (i.e., IH server address, port, query text, etc.) and checking that the values are within certain constraints (e.g., the address is a valid TCP/IP address, the port number is non-negative, etc.). After validation, the gateway identifies the action being requested by the user and performs the specified action. For some actions (e.g., query, expand, show) the IH server is contacted for the desired information. For others, the gateway can handle the request itself. In cases where interaction with the IH server is necessary, the gateway determines the response type for the IH server and performs the necessary reformatting of any returned data. The gateway converts the response into an HTTP compliant message and ships it back to the HTTP browser.

The gateway supports a number of different "actions" that a HTTP browser can request. Each of these actions is described below.

A "setup" request presents the user with the initial IH

server setup screen. This screen is used to set default values used in other interactions with the gateway. This action is normally the first action in a set of interactions between the user and the gateway.

The "init" request determines the host name of the IH server, the port where the server is accepting requests, and the DISPLAY value of the user's machine. Default values for these variables are maintained in the gateway and are presented to the user. The end user may alter any of these values from the setup screen. The values submitted by the user are then maintained across invocations of the gateway by adding them to all URLs created by the gateway and returned to the user. Once the user has specified these values and has submitted the request to the gateway, they are presented with the list of collections that the IH server they specified can access.

The "expand" request expands collections. Expanding a collection has a different meaning for different types of collections. For indexed (i.e., searchable) collections, expand provides a form-based interface for specifying search arguments for the collection. For all other collections, expand causes a request to be sent to the IH server asking for a particular IH collection (specified by an object ID). The results of this request are formatted in HTML for display back to the HTTP browser. The HTML will not include a URL to the parent collection when the object's type is LIST; otherwise, a URL to the parent will be included in the HTML.

A "query" request performs a query on an indexed

collection. The query text is passed to the IH server and if the collection contains any information units that satisfy the search criteria, the IH server returns a list of the IHO IDs corresponding to the information units. If no matching information units were found, the IH server returns a message stating that no matches were found.

The "show" request provides the user with a capability to view particular object. The object ID of the desired object and the HTTP browser's DISPLAY value are passed to the IH server. The IH server will either return the desired object to the gateway (which then passes the object back to the HTTP browser), or it will start a process to display the object back to the HTTP browser.

E. DESCRIPTION OF THE IH SERVER

The IH Server is key to our inventive system and provides the end-users with access to a set of IH Objects (IHOs) that make up that server's repository. Upon start-up, the server is told what collections will make up that server's repository. For each collection specified, the server locates, reads, and parses the collection's metadata file, constructing an internal (in-memory) representation of the IHOs and their relationships. Each IHO in memory is an instance of an "artifact" C++ subclass; the particular subclass depends upon the type of the IHO and determines how the object will handle incoming HTTP browser requests. Once it has read the metadata, the server goes into an event loop where it waits for incoming requests from the Gateway, processes those requests, and sends back appropriate

responses.

The following sections describe the processing performed by the server in more detail.

The IH server is initialized either manually by an administrator or automatically during a machine's boot cycle. The server is told which collections will make up its repository through various command-line arguments. For each collection, an **ihMeta** object is constructed to read and parse the metadata for that collection (see table 75 in Figure 6). Each collection is stored in its own subdirectory and contains a file called IH_SUMMARY that contains meta-information about the collection. The server uses that meta-information to determine specifically which IHO metadata files to read.

Each metadata file contains entities describing encapsulated IHOs and their inter-relationships. The **ihMeta** object parses each entity one at a time. An entity can be either an IHO or a relationship. For each IHO entity, a new **ihArtifact** C++ object is constructed. The object is actually an instance of one of the concrete classes derived from **ihArtifact**. The particular concrete class generated depends on the IHO's type attribute; each artifact subclass defines specific behavior for various requests against that type of object. The type thus determines how the artifact will respond to end-user actions on the object. Once the object has been created, it is added to a global object table for future reference, using the ObjectId as the key.

Relationship entities designate parent-child associations between two objects. When a relationship is read from the

metadata file, the server looks up both "ends" of the relationship in a global object table and establishes a bi-directional reference between the parent and child artifacts (i.e., the child is added to the parent's set of children and the parent is added to the child's set of parents).

While parsing metadata, if the ihMeta object detects malformed entities it reports appropriate error messages to the administrator. If too many errors are found, the server aborts before reaching the event loop.

Once the server has successfully read in all of its collections, it goes into the main event loop and waits for requests from clients.

The IH server runtime object model is based upon a class hierarchy of abstract and concrete C++ classes. Every IH Object has both a type and a subtype. The type defines which concrete class will represent the IHO in the server's internal representation of the object and how, in general, the object will respond to user actions. The subtype determines how those general actions on the object will actually be implemented (for instance, server-side PostScript objects (type MM, subtype postscript) get displayed by running Ghostview while server-side FrameMaker objects (type MM, subtype frame) get displayed by running FrameMaker software. The types and subtypes of the objects are determined by the extractors during collection preparation.

Figure 7 shows a class inheritance diagram for the ihArtifact family of classes. ihArtifact is an abstract class that defines the

interface to all IH Objects in the system. As an example, the ihArtifact abstract class 80 inherits the attributes from the ihArtFile objects 82 and the inArtSet objects 84.

Figure 8 depicts a table that defines the abstract interface to artifact objects. Figure 9 depicts a table containing descriptions of how each of the subclasses implements those methods described in Figure 8.

Each metadata entity in a repository is represented at runtime by an instance of a class in the ihArtifact hierarchy. These artifacts are maintained via two mechanisms: (1) an object table that maps object IDs to artifacts, and (2) a graph, linking objects by two-way parent-child relationships. As the metadata entries are read from files and instantiated as artifacts, they are added to the object table. This table is stored in an instance of the ihGraph class (see Figure 10) called "graph". Figure 11 shows an example of the primary object relationships in the server at runtime.

Once the server has finished loading all of the metadata from the repository's collections, the server enters the main event loop. The main loop is responsible for reading and processing requests. In pseudocode:

```
Do forever:
Wait for an incoming connection from a client
Spawn a new process to handle the request(s)
    For each incoming request (normally only one),
           Read the request
           Process the request
           Return the response to the client
    Close connection and exit child process
```

The server processes each incoming request as it is

received from the HTTP browser. The server contains a global instance of the class ihlpc called "server" that handles the inter-process communications. The main event loop asks the "server" object to read the next request; once read, the request is passed on to the metadata graph object for processing. The graph parses the request to determine the object ID of the object being acted on as well as the action to take on it. The graph looks up the artifact in its object mapping table, invokes the appropriate method on that artifact, and captures the results. The results are then returned back to the HTTP browser. Figure 12 shows an example of this behavior in an object interaction diagram. The main event loop 100 tells the server object 101 to read a request and tells the graph to process 102 the request. The graph invokes the appropriate method on the artifact (in this case, activate 103), which may in turn runs a browser script 104 to actually retrieve the desired data. The results are returned to the gateway by the server object.

Each object type in the IH server responds to user interactions in its own way. Sometimes this functionality is coded directly in C++ in the IH server, other times the functionality is dependent upon "helper" programs called "browser-scripts." A browser-script defines type/subtype-specific mechanisms for accessing an object.

The input to a browser-script is a location parameter that identifies the object to be viewed. The responsibility of the browser-script is to display this object to the user; how this is achieved depends upon the kind of data contained in the

object and how that data is to be shown to the user. For example, the browser-script for PostScript documents is invoked when the user wants to display a document whose type is MM ("server-side" multimedia) and whose subtype is ps. The PostScript browser-script takes the name of a PostScript document and executes a viewer program (i.e., ghostview) to display that document. The C browser-script is passed the name of a C file and the name of a function within that file; the script extracts the specified function and sends that text back to the invoking program (the server).

There are two implementation details that are not central our invention but which are important to highlight in this embodiment: (1) The encapsulate method for executing system commands; and, (2) the ihBlockMgr class for capturing large output.

There are several instances where the IH server needs to execute a UNIX program (such as a Perl script) and capture its output. For example, the server runs Perl programs called "Browser-scripts;" these scripts display the contents of an object to the user in a type- and subtype-specific manner. Additionally, when the server queries an index, it needs to run an indexer-specific Perl program, which in turn executes a search program and formats the responses. The stand-alone function "encapsulate" is used for both of these tasks.

Encapsulate forks a new child process and establishes the equivalent of a pipe between the parent and child processes: the child's standard error and output are redirected back to the

parent, which then reads that output. The output from the child is collected in a dynamically sized buffer (see the Block Manager, below); the buffer can then be sent back to the HTTP browser if necessary.

The GNU String class is not sufficient by itself as a data structure for storing arbitrarily long byte streams because it is restricted to containing a maximum of about 32,768 bytes. Therefore, a more sophisticated mechanism is required for capturing the output of browser scripts or for reading in arbitrarily large files. The ihBlockMgr class serves this purpose. This class maintains a sequence of zero or more "blocks," or buffers, of data. Each block can hold up to a fixed number of bytes. As data is being captured by the encapsulate function or read in from a file, it is written into the last block in the block manager's sequence. When the current block fills up, a new block is added to the sequence. Thus, the block manager is an efficient way to hold a dynamically growing stream of bytes. In addition to providing mechanisms to add data to the block manager (which is instantiated once globally), ihBlockMgr includes methods for iterating through the blocks one at a time and for clearing out the manager's contents.

E. SUBSYSTEM INTERACTION

Within our pre-processing methodology we define a process "in_prep", which is a Perl script used to extract metadata. In_prep cooperates with two other types of programs: extractors and indexers. Extractors are type specific Perl subroutines required by in_prep to traverse physical data and extract the

necessary information required for metadata and indexes. A separate extractor is needed for each type of data placed under control of an IH server. Indexers can be implemented using any language desirable. The only limitation imposed is that the in_prep process must be able to access the indexer via the Perl "system()" function. Indexers are not type specific, since they can be applied to any text data. Indexers are used to provide content-oriented queries over physical data. Figure 13 illustrates the interaction that take place between in_prep, extractors, and indexers. For each invocation of in_prep 111, an extractor is called to process each member of the desired information units. The in_prep process passes the location of the physical data (usually a file name) to the extractor 112. The extractor in turn processes the physical data (referred to as an information unit IU) and extracts metadata as well as text to be indexed from the IU, and if there is more than one IHO in the IU, the extractor also establishes relationships between the objects.

The objects and relationships created by the extractor 112 are returned to in_prep 111 which writes them to the metabase for use later by the IH server.

In_prep 111 invokes the appropriate indexer to index 113 the text data extracted from the IU. The output of the indexer is saved in the metabase for later use by the IH server.

The metadata entries produced by in_prep and stored in the metabase are loaded into memory by the IH server at run time. The IH server then enters a loop where it responds to

incoming requests from HTTP browsers. Referring back to Fig. 3, after the server is initialized and running, the IH server enters a main event loop and waits for requests from clients 38. End-users then access the IH server through an HTTP server 40. Once the end-users access the IH server, they perform one of three actions to select an object 42: (1) a metadata based query, (2) a content based query, or (3) explicitly navigate around the IHOs. Once an object is selected, it can be accessed and browsed by activating either a client side browser 44 or server side browser 46. The user may also operate on the object choosing from a set of procedures such as print, store, fax, etc.

Figure 14 illustrates the processing of a request by an end-user for conducting a metadata query. A client requests 121, via HTTP, the initial collection held by an IH server. The request is passed, via the CGI 122, to the gateway. The gateway connects to the IH server and requests 123 the initial collection via a internal protocol. The IH server determines the initial collection based upon its in-memory metadata and returns the results to the gateway 124. The gateway reformats the response into HTML and sends 125 its response to the HTTP server. The HTTP server passes 126 the results back to the HTTP browser client without interruption since our gateway is a "no parse header" gateway. This means that the HTTP server will do no parsing of our response, and the gateway must be able to form correct HTTP responses.

Figure 15 illustrates the process for conducting a

context-oriented query. The end-user via HTTP, for an InfoHarness collection held by an ih_server requests a context-oriented query 151. The request is passed via the CGI to the gateway 152. The gateway connects to the ih_server and requests a context-oriented query 153, passing the query text. Based upon the type of the InfoHarness collection, the proper indexer is invoked to perform the search 154. The indexer returns a list of IHOs that satisfy the query 155. The IH server returns the list of IHOs to the gateway 156. The gateway reformats the list of InfoHarness objects in the HTML and returns the list to the HTTP server 157. The HTTP server transmits the list of objects to the HTTP browser 158.

Figure 16 illustrates a the processing of a request for invoking a server side browser. A client requests, via HTTP, an IH object held by an IH server 161. The request is passed via the CGI to the gateway 162. The gateway connects to the IH server and requests the IH object via any internal protocol 163. IH server determines that the requested object requires the invocation of a server side browser 164. The correct browser is invoked with the location of the object. The browser starts a process that displays the object back to the client's machine 164. Any error text generated by the browser is returned to IH server 166. IH server returns a message to the gateway indicating either successful invocation of the browser, or error text generated by the browser 167. If an error message was received from IH server, it is reformatted into HTML and passed back to the HTTP server 168, otherwise, the gateway indicates

success via the HTTP 169 OK message. The response from the gateway is transmitted to the user via HTTP 170. As long as the user does not close the application started by the browser, they can invoke any actions supported by the application and the results will be sent back to the machine where the browser was started. (Note the security risks associated with server side browsers. The user has access to an application that runs with the inherited permissions of IH server. This implies that the user may be able to open other files, change other files, and may even be able to escape to the shell on the machine where the browser was started (again inheriting the identity of the user that started IH server).

Figure 17 illustrates the process for a request for invoking a client side browser. A client requests via HTTP, to see an IH object held by an IH server 171. The request is passed via the CGI to the gateway 172. The gateway connects to the IH server and requests the IH object 173. IH server examines the type of the object requested and determines that the object can be displayed using a client side browser (or in HTTP browser terms, an external viewer). The location of the object is determined and the IH server returns the contents of the file to the gateway 174. The gateway performs a mapping between the IH subtype of the object and the MIME type corresponding to the object. This MIME type is returned with the object contents to the HTTP server 175. The HTTP browser receives the contents of the object and determines which external viewer to invoke for the specified MIME type 176. The contents of the object are

stored in a temporary file. The external viewer is started 177 with the name of a temporary file that contains the contents of the requested object.

It is to be understood that the method and system for providing uniform access to heterogeneous information as illustrated herein are not limited to the specific forms disclosed and illustrated, but may assume other embodiments limited only by the scope of the appended claims.

We Claim:

1. A system for providing uniform access heterogeneous data from a plurality of end-users, said system comprising:

a database of metadata extracted from a plurality of information sources; and

a server having loaded in memory, instantiations of said metadata from said database

2. The system as claimed in claim 1 wherein further comprising information servers containing said information sources connected to said server.

3. The system as claimed in claim 2 further comprising a plurality of end-users operating HTTP compatible browsers all connected to said server.

4. The system as claimed in claim 3 wherein said instantiations of said metadata loaded in said server memory are organized into objects, collections, and respositories.

5. A method for providing a plurality of end-users access to individual information units of heterogeneous information, said method comprising:

pre-processing said individual information units of heterogenous information to extract metadata for each of said informtion units;

creating a database of said metadata;

loading said metadata from said database into a server's resident memory;

placing said server into a main line loop awaiting requests from said end-users;

receiving requests for information at said server from said end-users; and

responding to said requests using said metadata

stored in said resident memory.

6. The method as recited in claim 5 wherein said database created from said metadata organizes said metadata into objects and collections.

7. The method as recited in claim 6 wherein the step of loading said metadata from said database includes the steps of loading said objects and collections, and further includes the step of organizing said objects and collections into repositories.

8. The method as recited in claim 6 wherein said request received from said end-users is either a metadata query or a information content query and wherein said server responds to said query returning one of said objects satisfying said query.

9. The method as recited in claim 8 wherein said responding step further includes the step of invoking a client side browser to view said information units identified by said one of said objects.

10. The method as recited in claim 8 wherein said responding step further includes the step of invoking a server side browser to view said information units identified by said one of said objects.

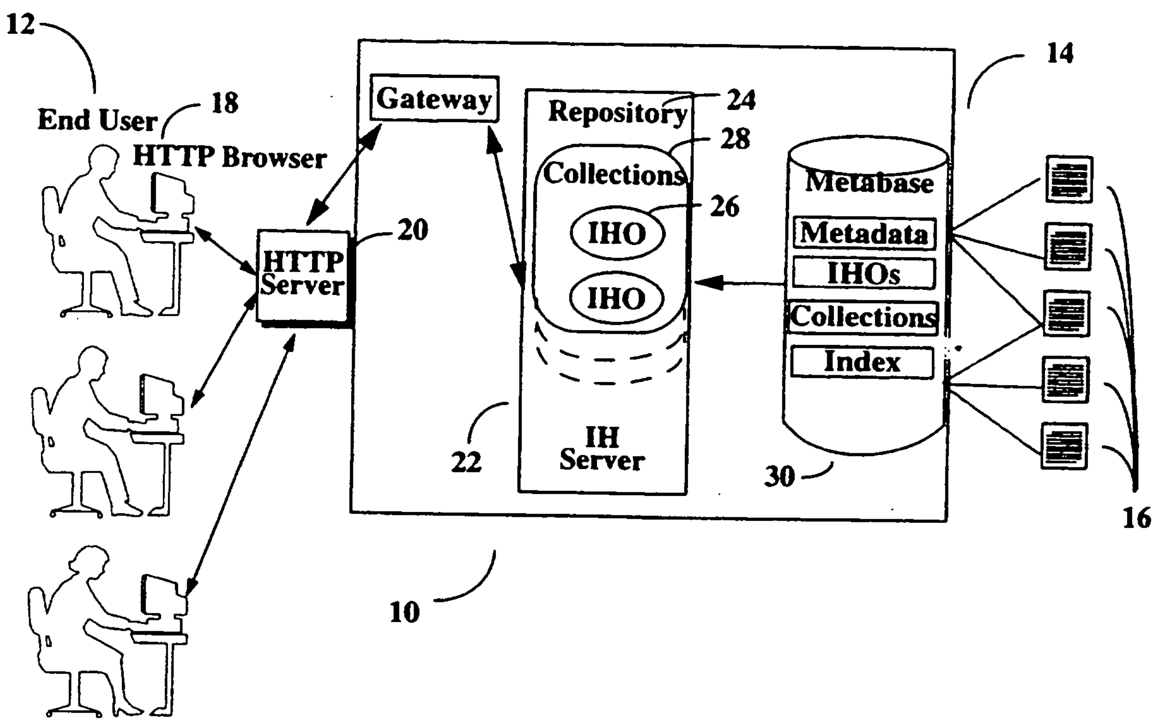

12
End User
18
HTTP Browser
20
HTTP Server
Gateway
Repository
24
28
Collections
26
IHO
IHO
22
IH Server
14
Metabase
Metadata
IHOs
Collections
Index
30
16
10

Fig. 1

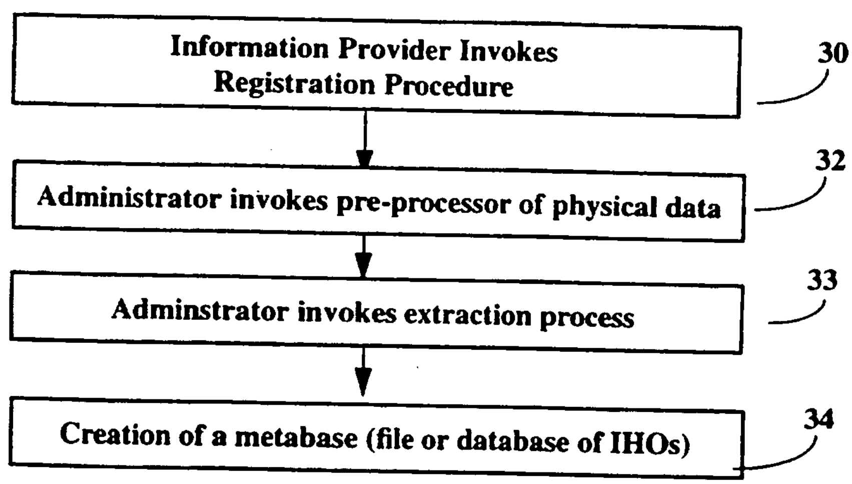

Information Provider Invokes Registration Procedure
30
Administrator invokes pre-processor of physical data
32
Adminstrator invokes extraction process
33
Creation of a metabase (file or database of IHOs)
34

Fig. 2

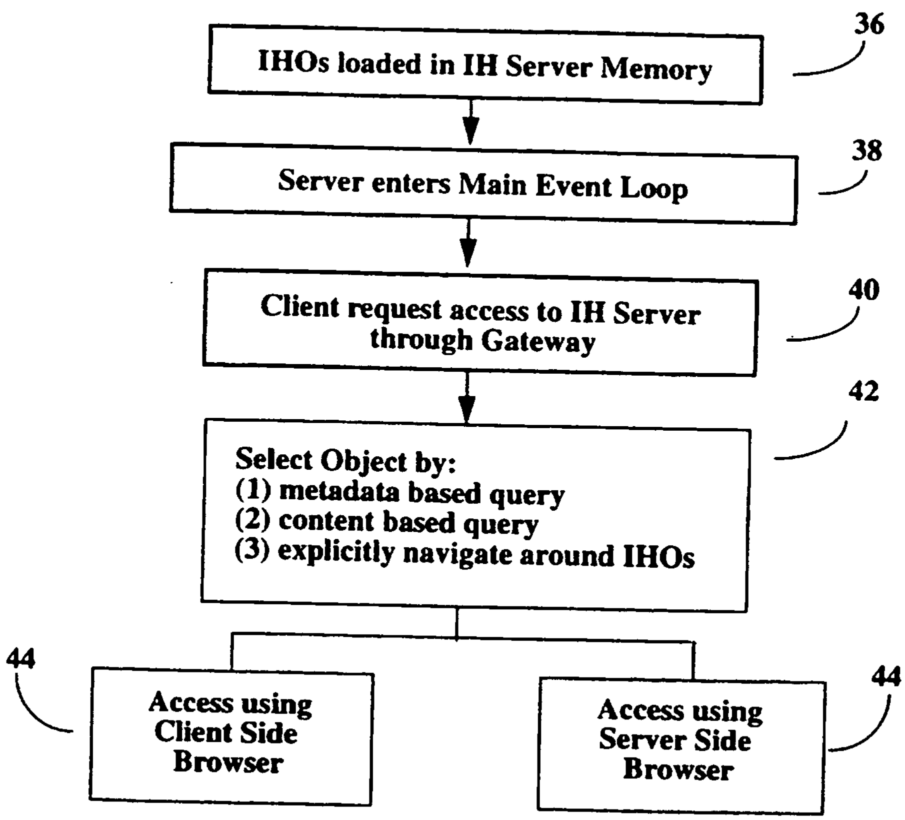

IHOs loaded in IH Server Memory
36
Server enters Main Event Loop
38
Client request access to IH Server through Gateway
40
Select Object by:
(1) metadata based query
(2) content based query
(3) explicitly navigate around IHOs
42
Access using Client Side Browser
44
Access using Server Side Browser
44

Fig. 3

48 52

f11:f12:f13:f14:f15:f16|f21:f22:f23:f24:f25:f26| ...

50

Fig. 4(a)

Field Number	Name	Description
1	Location	Indicates where the IU is stored so that it can be retrieved later
2	Unique ObjId Indicator	Whether ih_prep should use the location to construct a unique object ID
3	Depth	Used to create parent-child relationships between IUs
4	Subtype	Subtype for the object (e.g., frame, ps)
5	Subject	Summary information displayed to the user during browsing
6	Text	The extracted text from the IU, used if necessary during indexing

55 56 58 60 62 64 54

Fig. 4(b)

```
(objid|type|subtype|location|attribute=value|attribute=value..)
```

70 71 72 73 74

Fig. 5

ihMeta Class

Description	Reads and parses the metadata entries for a collection, constructing artifact objects and establishing relationships between them as the data is read.
Primary Methods	• `readMetadata`: Static method called by main module to read metadata from a file. • `getNextKind`: Determines the next "kind" of metadata entity (object/relationship). • `getArtifact`: Reads and constructs the next artifact object. • `getRelationship`: Reads the next relationship.
Primary Collaborations	— *ihGraph* (uses) — *ihArtifact* (constructs) — `main()` (used by)

75

Fig. 6

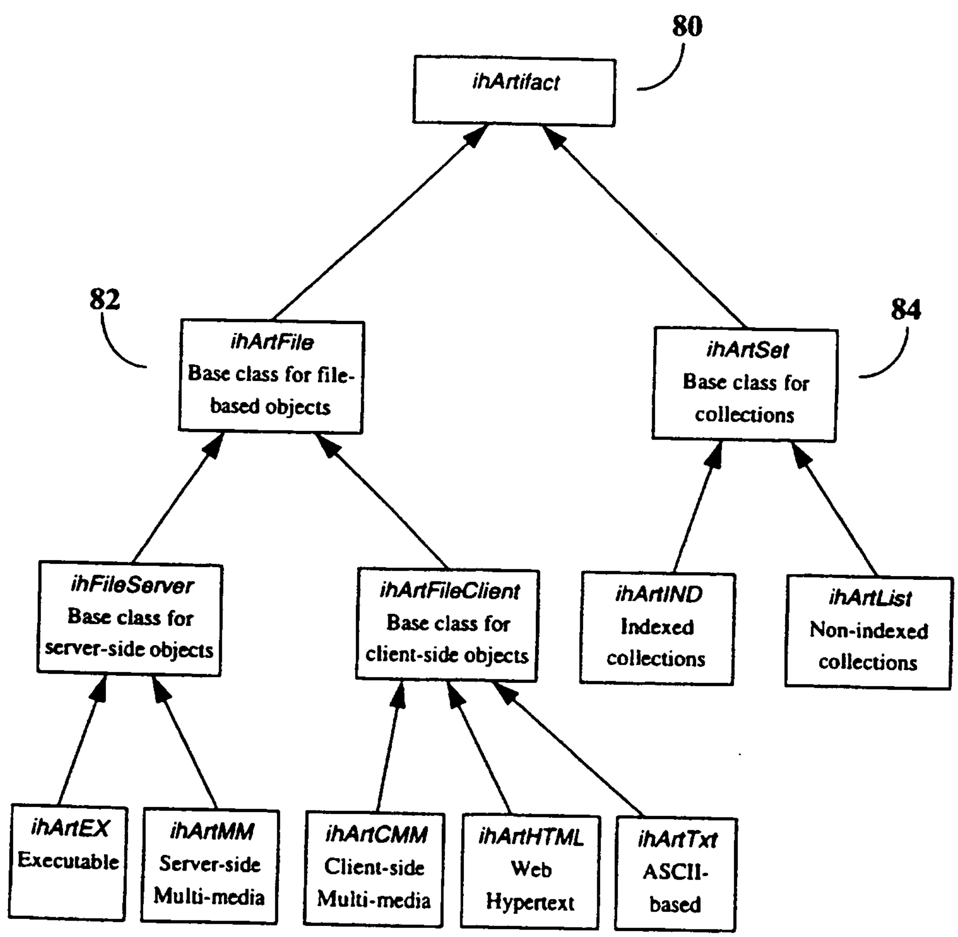

80
ihArtifact
82
ihArtFile
Base class for file-based objects
84
ihArtSet
Base class for collections
ihFileServer
Base class for server-side objects
ihArtFileClient
Base class for client-side objects
ihArtIND
Indexed collections
ihArtList
Non-indexed collections
ihArtEX
Executable
ihArtMM
Server-side Multi-media
ihArtCMM
Client-side Multi-media
ihArtHTML
Web Hypertext
ihArtTxt
ASCII-based

Fig. 7

ihArtifact Class

Description	Defines abstract interface to artifact objects, which are in-memory instantiations of the IOs encapsulated by metadata. These objects respond to actions taken by the end-user during navigation through "IH-space."
Primary Methods	• `getAttribute`: Returns the value of an attribute. • `getType`: Returns the type of the artifact. • `getParent/getChild`: Returns an artifact's parent or child given an *ObjectId*. • `format`: Creates a "flattened" ASCII representation of the metadata in the artifact, including attribute/value pairs and numbers of parents/children. • `formatChildren`: Creates a string of the artifact's children, flattened by `format`. • `display`: View the object encapsulated by the artifact. The following are directly invoked by client requests: • `query`: For indexed collections, issue a query against the collection and return the matching objects. • `expand`: Return a representation of the object (typically via format). • `activate`: Perform appropriate action when user "clicks" on the object (e.g., view the object by calling `display`).
Primary Collaborations	— *ihGraph* (managed by) — *ihMeta* (constructed by)

Fig. 8

ihArtifact Subclasses -- How they Override Key Methods

Class	query (QUERY)	expand (EXPAND)	activate (SHOW)
ihArtFile	ERROR	Return formatted object and its children.	Invokes virtual display method, which is overridden in derived classes to "show" the object. (Descriptions below describe how display works).
ihArtFileClient	ERROR[a]	Return formatted object and its children.[a]	For subtype none, returns contents of file. Otherwise, runs browser script and returns its output.
ihArtFileServer	ERROR[a]	Return formatted object and its children.[a]	N/A (pure virtual)
ihArtTxt	ERROR[a]	Return formatted object and its children.[a]	For subtype none, returns contents of file. Otherwise, runs browser script and returns its output.[a]
ihArtHTML	ERROR[a]	Return formatted object and its children.[a]	For subtype none, returns contents of file. Otherwise, runs browser script and returns its output.[a]
ihArtMM	ERROR[a]	Return formatted object and its children.[a]	Run the browser script to display the object (usually over X connection) and return the results.
ihArtCMM	ERROR[a]	Return formatted object and its children.[a]	Return the contents of the file; Gateway will add MIME-type to header so client will view object.
ihArtEX[b]	ERROR[a]	Return formatted object and its children.[a]	Run the executable for the object's subtype, returning the results.
ihArtSet	ERROR[a]	Return formatted object *without* its children.	Return formatted object *without* its children.
ihArtList	Returns the entire list of objects.	Return formatted object and a list of children, making sure returned results won't overflow size limits.	Return formatted object and a list of children, making sure returned results won't overflow size limits.
ihArtIND	Issues a query against the indexed collection and returns list of matching objects.	Return formatted object *without* its children.[a]	Return formatted object *without* its children.[a]

a. Inherited method from parent class.

Fig. 9

ihGraph Class

Description	Maintains map of object IDs to artifact objects; processes incoming requests by passing them to appropriate artifact.
Primary Methods	• `getEntry`: Returns the "top-level" object, the collection without any parents. • `getArtifact`: Returns an artifact object given its object ID • `process`: Process an incoming request from the client.
Primary Collaborations	— *ihMeta* (modified by) — `main()` (constructed by, used by)

Fig. 10

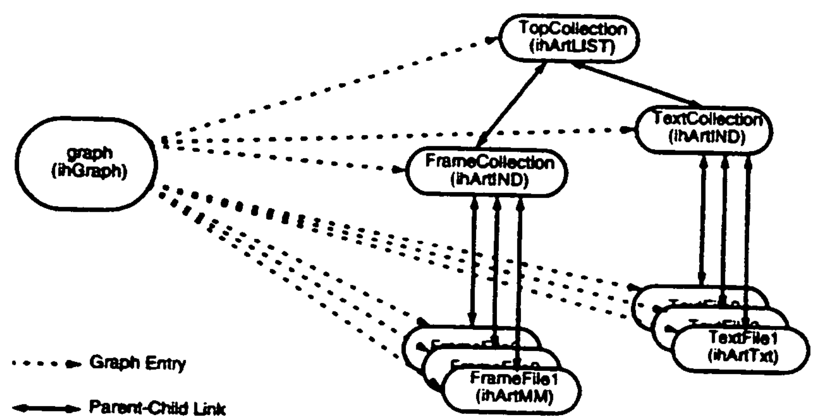


Fig. 11

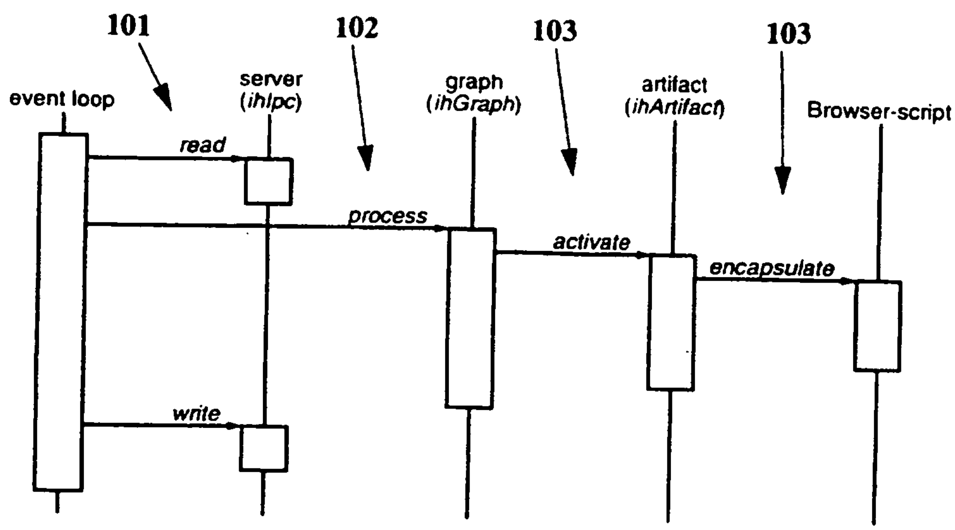

101
102
103
103
event loop
server
(ihIpc)
graph
(ihGraph)
artifact
(ihArtifact)
Browser-script
read
process
activate
encapsulate
write

Fig. 12

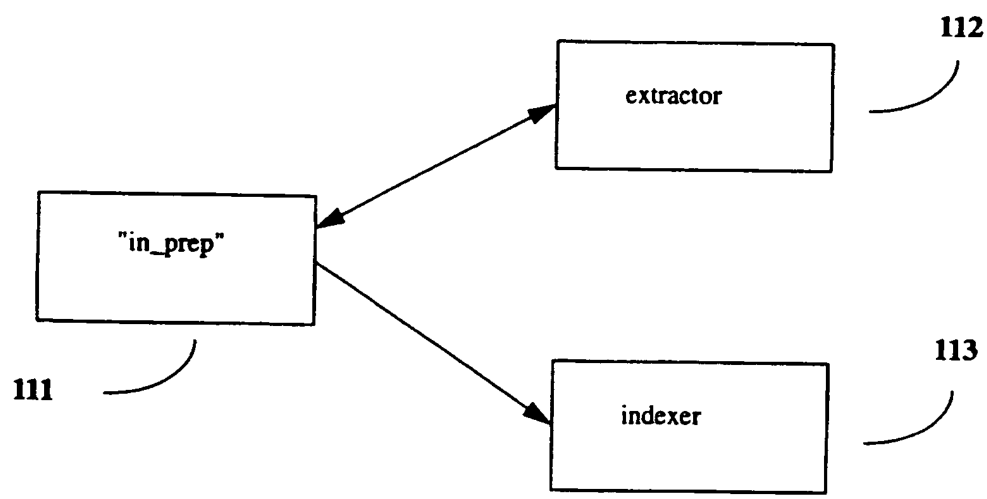

112
extractor
"in_prep"
111
113
indexer

Fig. 13

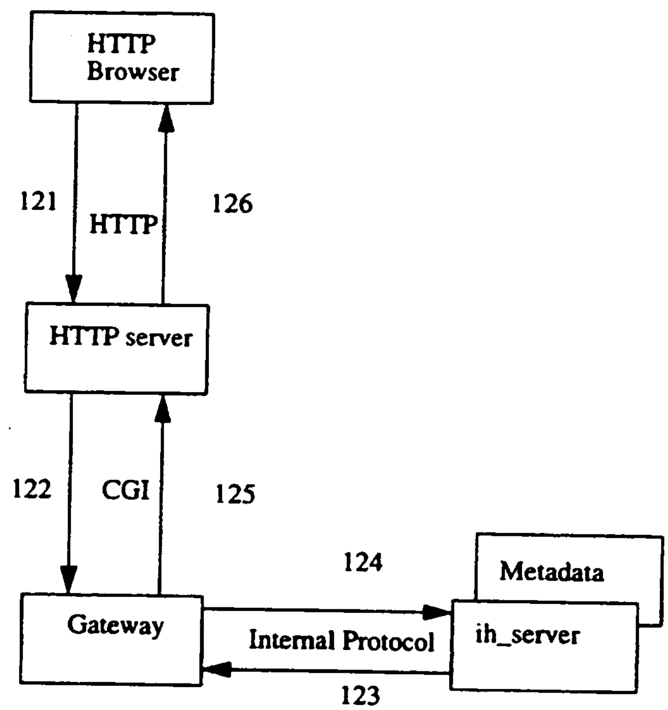

HTTP
Browser
121
HTTP
126
HTTP server
122
CGI
125
124
Metadata
Gateway
Internal Protocol
ih_server
123

Fig. 14

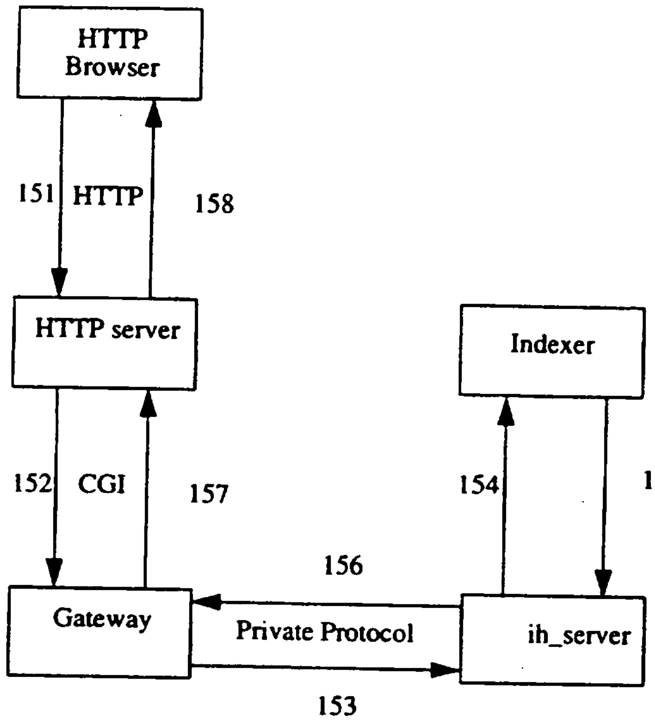

HTTP Browser
151
HTTP
158
HTTP server
152
CGI
157
Indexer
154
155
156
Gateway
Private Protocol
ih_server
153

Fig. 15

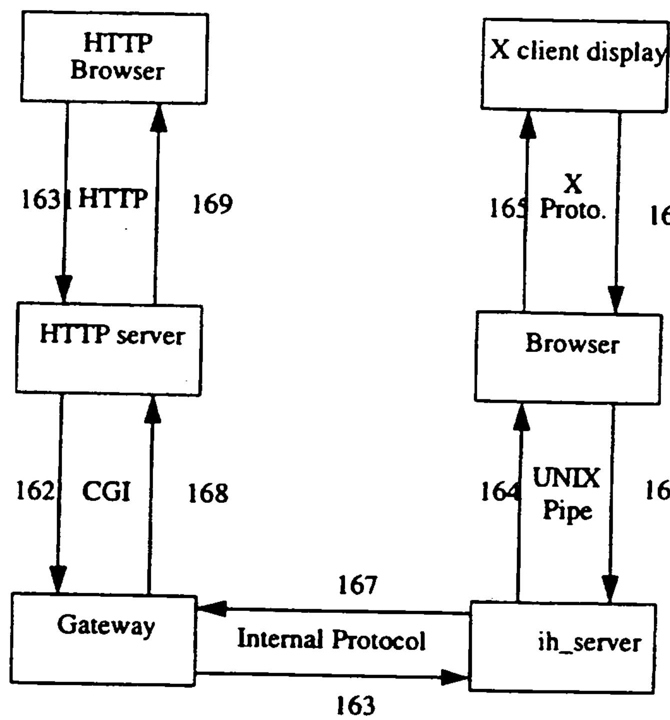

HTTP Browser
X client display
163
HTTP
169
X Proto.
165
160
HTTP server
Browser
162
CGI
168
164
UNIX Pipe
166
167
Gateway
Internal Protocol
ih_server
163

Fig. 16

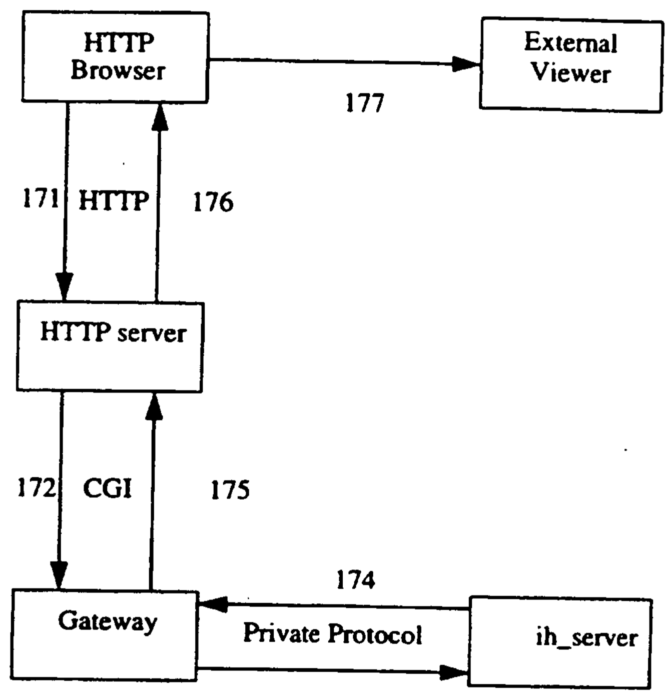

HTTP Browser
External Viewer
177
171
HTTP
176
HTTP server
172
CGI
175
174
Gateway
Private Protocol
ih_server

Fig. 17

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US96/15620

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) :G06F 17/30
US CL : 395/601,604,610,200.09
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 395/601,604,610,200.09

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

APS, DIALOG (PATENTS, COMPSCI, 35), COMPUTER SELECT, PROQUEST, INTERNET

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X,P ---------- Y,P	US 5,493,677 A [BALOGH ET AL.] 20 February 1996, see entire document.	1-3 ------ 4-10
A	Slonim et al., The Information Utility: a Project Retrospective, Software Engineering Journal, Vol: J5, No: 4, July 1990, pp. 223-236.	1-10
X ---------- Y	Freeman et al., Hosting Services- Linking the Information Warehouse to the Information Consumer, Digest of Papers. Spring COMPCON 94, San Francisco, California, 28 February - 04 March 1994, pp.165-171, especially p 165, col 2, p. 166, col 1,p. 167 col. 2, Figure 1, Figure 2.	1-2,5 -------------- 3,4,6-10

[X] Further documents are listed in the continuation of Box C. [ ] See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search 18 NOVEMBER 1996	Date of mailing of the international search report 13 DEC 1996
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. (703) 305-3230	Authorized officer PAUL R. LINTZ Joni Hill Telephone No. (703) 305-3832

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	Robert Hess, "Cyberdog to Fetch Internet Resources for OpenDoc Apps.", MacWEEK, Vol: 8, No: 44, 07 November 1994, pp 1-2., especially page 1, lines 13-22	1-10
A	David Stodder, Reinventing the Database: Data Warehouse and Economics will Shift the Database Landscape in 1995, Database Programming and Design, vol. 8, no. 1, January 1995, pp. 7-9.	1-10
A	Colin White, The Key to a Data Warehouse: Unlocking the Secrets of Data Warehousing With thye Information Directory, Database Programming and Design, vol. 8, no. 2, February 1995, pp 23-265.	1-10
A	Ellis Booker, US West Chanpions Internal Internet, Computerworld, vol. 29, no. 11, 13 March 1995, pp. 2.	1-10
Y	R. Oswald, Manitoba Land Related Information System: The Information Utility, IEEE WESCANEX 95: Communications, Power, and Computing Conference Proceedings. Winnipeg, Manitoba, Canada, 15-16 May 1995, pp. 252-257, especially, p252, 253 column 1, Figure 1, p. 255, Cols. 1 & 2.	1-10
A	George A. Thompson, Warehouse? There House!, HP Professional, vol. 9, no. 5, May 1995, pp11-13.	1-10
A	Amy Rogers, Oracle Intros Add-Ons for Data Warehouses, Communications Week, no. 567, 24 July 1995, pp. 15-16	1-10
A	C. James, What goes into an Information Warehouse?, Computer, vol. 28, no. 8, August 1995, pp. 84-85,	1-10